DFAs in OCAML

We describe here our implementation of DFAs in OCAML. The files that implement this are in the ocaml folder, namely dfa.ml and dfa.mli.

Let's start by the interface:

```
exception InvalidDFA

module type DFA =
   sig
      module A : Alphabet.A
      type state = int
      type elem
      type str
      (* The dfa type *)
      type t

      val make : int -> (state -> elem -> state) -> state list -> t

      val delta : t -> state -> elem -> state
      val deltaStar : t -> state -> str -> state
      val accept : t -> str -> bool

      (* Returns the accepted strings of at most given length *)
      val acceptedStrings : t -> int -> str list

      val union : t -> t -> t
      val intersect : t -> t -> t
   end

module Make(A : Alphabet.A) : DFA with type elem = A.elem
                                   and type str = A.t
```

The module type DFA describes a DFA on a prescribed alphabet A. It introduces a number of types: one for states, represented simply as integers, one for elements of the alphabet, one for strings from the alphabet, and finally a type t to represent a dfa.

The first key method is make, which creates a new dfa. It takes 3 inputs: First the number of states, then a transition function that given a "state" and an element returns a new state, and finally a list of "final states". It returns a dfa (doing some validation first). By convention, the state corresponding to the number 0 is automatically treated as the start state, so no need to specify it.

Following are methods allowing us to trace the accepting of strings: delta carries out one step of the transition function, deltaStar carries out a whole sequence of steps, and accept determines whether the string is accepted by the dfa.

Lastly, acceptedStrings returns all strings of length up to a given integer that are accepted by the dfa.

Finally, two methods implement the construction of the union and intersection of dfas, that we will be discussing in class.

The implementation of dfa.ml is for the most part straightforward. We represent dfas as a *record type*, which we haven't talked about before but should be straightforward:

```
type t = {
    nstates : state;
    delta : state -> elem -> state;
    final : state list;
}
```

The function make essentially just wraps its 3 arguments into an object of type t, and validates it first before returning it (to make sure that the transition function does not take you out of the valid state range, for example, and that the valid states are actually valid states).

The delta function literally just returns the value stored in delta. It is worth noting this expression:

```
let delta { nstates; delta; final } = delta
```

The part { nstates; delta; final } is basically a pattern matching a record. It would normally be written as: { nstates = nstates; delta = delta; final = final } where we use the field names also as variable names. The above is a shorthand for that.

Next up is deltaStar, which is supposed to follow the transition function through a list of inputs. A simple List.fold_left does this nicely.

Then here is accept:

```
let accept ({ nstates; delta; final } as dfa) es =
    List.mem (deltaStar dfa 0 es) final
```

Note the expression ({ nstates; delta; final } as dfa). This says that the first argument should match a record, and bind the 3 arguments to the variables nstates, delta and final, but that the whole argument should also be bound to the variable dfa. All the function does then is use deltaStar to follow the string's steps, starting from the start state 0, and check whether the resulting state is one of the final states.

Lastly, acceptedStrings generates all lists up to a given length, using A.allStringsLeq, then uses List.filter to only keep those that pass the dfa's accept.

Union of regular languages

In another section we saw that the union of two regular languages is regular, by explicitly constructing, using the two DFAs for the two languages, a new DFA that would accept their union. The idea back then was that the states of the new DFA would be pairs of states from the old DFA. We will now carry out the construction.

The first thing we have to worry about is the fact that for us the states are simply integers, starting at 0. We have no way to change that. So what we need to do is somehow associate pairs of numbers with a single number, in a unique way. There is sort of a standard way to do that:

Given n, m, define the functions:

$$f(N) = (N \bmod m, N \operatorname{div} m)$$

and

$$g(i,j) = i + m * j$$

Then these functions are inverses of each other and establish a 1-1 correspondence between the sets

$$\{0, 1, \ldots, nm-1\}$$

and

$$\{(i,j) \mid i = 0,1,2,\ldots,n-1, \quad j = 0,1,2,\ldots,m-1\}$$

Here the quantity N div m is integer division. So this way, since we denote the states of one DFA, say n of them, with the numbers 0 through $n-1$, and since we denote the states of the second DFA, say m of them, with the numbers 0 through $m-1$, the above functions show us how to associate the pairs (i,j) of states from the two DFAs into a single number in the range 0 to $nm-1$.

This is the key technical part of the construction of the DFA for the union. The rest is a bit of bookkeeping. Here is the code:

```
let union { nstates= nstate1; delta= delta1; final= final1 }
          { nstates= nstate2; delta= delta2; final= final2 } =
   let from12 i j = i + nstate2 * j in
   let to12 n = (n mod nstate2, n / nstate2) in
   let nstates = nstate1 * nstate2 in
   let newDelta n a = let (i, j) = to12 n
                      in from12 (delta1 i a) (delta2 j a) in
   let isFinal n =
      let (i, j) = to12 n
      in List.mem i final1 || List.mem j final2
   in {
      nstates= nstates;
      delta= newDelta;
      final= List.filter isFinal (upTo nstates);
   }
```

So let's take a look. The input to the function union is two records corresponding to the two DFAs. Because we need two of them and we'll use numbers to distinguish, we couldn't use the shorthand pattern notation, we had to spell all the equal signs out.

Now next we do a series of computations, via let statements, each feeding into the next, before a final let statement that returns our answer. The first is the computation of the function g above, which we call from12. Similarly a definition of the function f above, which we call to12. Next we compute the total number of states for the new DFA, and the new delta transition function, newDelta. Let's walk through how that works:

newDelta is given a number n corresponding to a state of the new DFA and an input a. It then uses to12 to compute from that n the pair of states it corresponds to. We store those states in the variables i and j respectively. Now we use the delta functions of the two DFAs to obtain a new pair of states, delta1 i a and delta2 j a, then we feed these back into the function from12 to get a single state number. That's the new transition function.

Next we need to determine the final states. If you recall, the final states were exactly those pairs where at least one of the states in the pair is a final state of the corresponding DFA. The function isFinal is meant to test exactly if a state in the new DFA is final.

Lastly, we can create the resulting DFA. It must be a record with 3 entries, the number of states (product), the new delta function (newDelta) and the list of final states, which we obtain by filtering all the states using the function isFinal. We use a little function upTo that produces the consecutive list of numbers [0, 1, 2, ..., n−1] for a given n.